5 Обработка текстовых данных

5.1 Текстовый формат представления данных

Текстовый формат представления информации – использование человекочитаемых (печатных) символов, образующих (так или иначе) «слова», «строки», «предложения».

Широкий диапазон применений обусловлен набором качеств:

– Естественность организации общении с пользователем-человеком – использование в интерфейсах пользователя (UI), но только диалоговых

– Удобство для контроля со стороны человека (оператора)

– Переносимость и совместимость при обработке и передаче (особенности каналов связи, коммуникационных протоколов, двоичных и низкоуровневых машинных форматов)

Недостатки:

– Дополнительные затраты на интерпретацию (как правило, машинная обработка числовых данных выполняется в машинном двоичном представлении)

– Дополнительные затраты на хранение и/или передачу (например, 5-6 байт вместо 2 для short int, 10-11 байт вместо 4 для long int)

Второй недостаток частично преодолевается, если для представления двоичной информации использовать не привычную текстовую запись, а запись шестнадцатиричными цифрами либо в кодировке Base64 или ей подобной. Но при этом ухудшается удобство чтения человеком.

В целом, для многих применений преимущества более весомы, что подтверждается, в частности, распространением XML в качестве универсального формата представления данных. В то же время, например, наличие многочисленных «компрессоров» XML «двоичных» альтернатив ему подтверждает также и актуальность проблем.

Разновидности текстовых форматов:

– «Обычный» ***неструктурированный*** («гладкий», ***plain***) текст – может быть разделен на ***строки*** и далее на ***слова***. Содержит специальные символы-разделители: «конец строки» («end-of-line» – *EOL*), «пробел» (и, возможно, другие «пробельные», например символ табуляции «`\t`»). Неструктурированный текст может содержать произвольную информацию, сколь угодно сложно организованную, но для ее извлечения потребуется отдельная процедура разбора – ***парсинг*** (***parsing***), при этом могут возникать неоднозначности, противоречия, неполнота данных и другие проблемы.

Отдельная проблема – представление конца строки. По историческим причинам на различных платформах для этого служат различные символы и их комбинации, чаще всего это «подача строки» («новая строка») и «возврат каретки»[1]:

Символ			**Интерпретация**
`ASCII 10`	**`0x0A`**	**`\n`**	«подача строки» – «line feed», *LF* «новая строка» – «new line», *NL*
`ASCII 13`	**`0x0D`**	**`\r`**	«возврат каретки» – «carriage return», *CR*

Эти же символы действуют и в кодировках Unicode: **`0x000A`**, **`0x000D`**.

[1] Названия исторические: в механических пишущих машинках движения рычага возвращали каретку к началу строки и затем перемещали бумагу на один интервал. Для машинок-консолей были введены соответствующие управляющие символы.

EOL	**Платформа**
LF	Multics; Unix и большинство Unix-подобных систем; современные Mac OS; …
CR	Commodore; ZX Spectrum; Apple II и ранние Mac OS; …
CR+LF	Atari TOS; Microsoft CPM/80, DOS, Windows; OS/2; Symbian, Palm OS; …

Различное представление «конца строки» встречается также в протоколах обмена данными, например сетевых. Программные реализации могут допускать использование нескольких вариантов *EOL*. Аналогично – текстовые редакторы

Системные библиотеки (библиотеки языка C) позволяют использовать обозначение «**`\n`**» независимо от платформы (только для файлов, открытых как текстовые): в скомпилированной программе будут подставлены корректные значения.

– Структурированный текст с ***разделителями***, или «***ASCII delimited***», например «space-delimited», «TAB-delimited», «Comma-separated» (CSV), и т.д. Фактически представляет собой таблицу (не обязательно прямоугольную). Разбор упрощается за счет лучшей формализации, но только для информации «табличного» характера.
Проблема экранирования символов-разделителей среди данных (***escapement***).

– Структурированный ***размеченный*** (***тегированный***) текст – структура данных задается специальными символами (***метасимволами***) и их комбинациями (в т.ч. фактически имеющими вид команд), а символы-разделители (все или некоторые из них) свое значение утрачивают.
Примеры: HTML, XML, TeX и т.д.
Универсальность, независимость от локальной интерпретации служебных символов, но проблема экранирования символов разметки и некоторых других в данных.

Возможен также смешанный подход, например plain-текст с отдельными тэгами в нем.

Примеры:

– Исходный текст программы на C – символы пробелов и конца строки безразличны, разбор по синтаксическим конструкциям.

– Текст скрипта Shell – конец строки разделяет «операторные строки»

– Текст сценария make (makefile) – учет отступов (пробелов или табуляций)

5.2 Базовые средства обработки текстов в Unix

В Unix-системах в силу исторических причин обработка текста (и данных, представленных в текстовой форме) занимает важное место.

Проблема – необходимость описывать текстовые «предложения» и выполнять достаточно широкий набор операций.

Решение – специализированный язык для описания элементов текста и операций с ними.

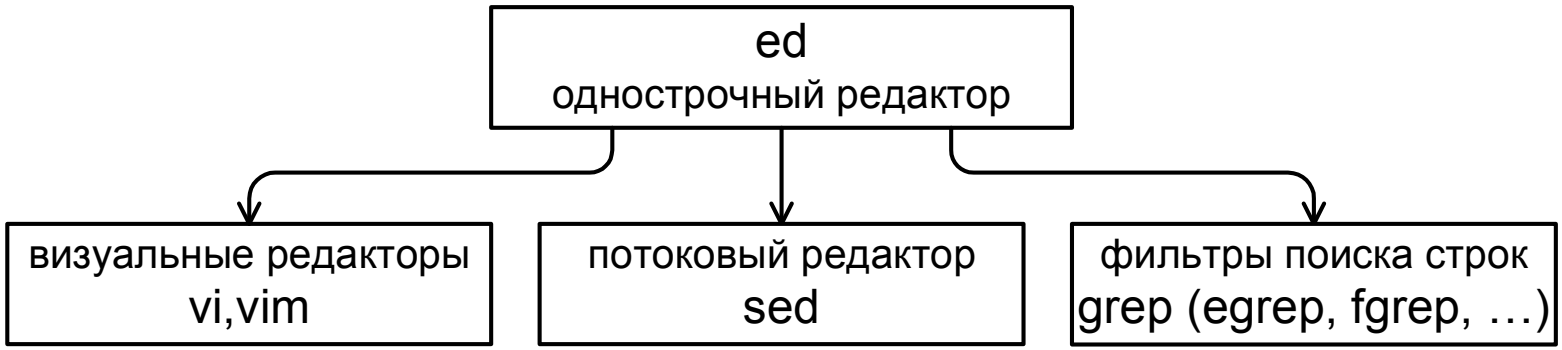
5.2.1 Однострочный редактор ed

Исторически первый ***стандартный*** текстовый редактор в Unix-системах – однострочный ***командный*** редактор *ed* (*ED*itor). Важная особенность: постоянное отображение редактируемого текста не предполагается, все действия над текстом (в т.ч. отображение) выполняются по командам, что было полезно при использовании в качестве консоли пишущей машинки, а также позволяло применять его в ***пакетном*** режиме.

Расширенная версия *ed* – *ex* (*EX*tended editor). Тоже однострочный, но существенно расширен набор команд для сложного поиска и модификации в тексте. Это множестве команд поддерживается и в редакторе *vi* (режим «расширенных» команд, *ex*-mode, см. ниже).

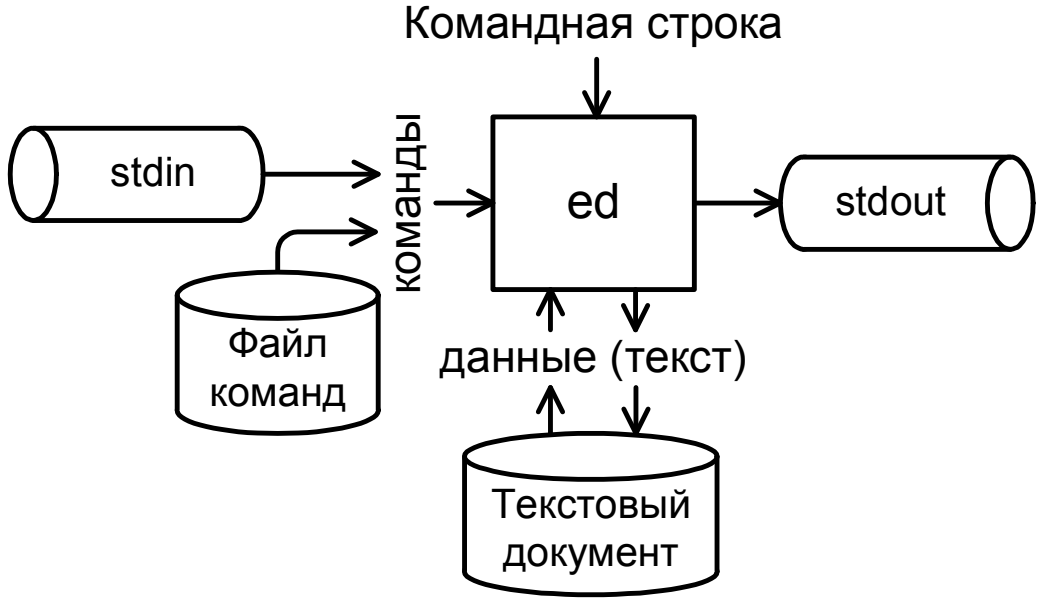
В современных системах *ed*, как правило, присутствует, а вместо «настоящего» *ex* команда **`ex`** запускает редактор *vi* (*vim*) в ex-mode.

Так или иначе, *ed* (и *ex*) сейчас почти не применяются, но они послужили основой для ряда современных утилит (точнее, их семейств): визуального редактора *vi*, потокового редактора *sed* и утилиты поиска в тексте или потоке *grep*.



Средства обработки текстов: *ed*, *vi*, *sed*, *grep*

Редактор *ed* нельзя считать в полной мере фильтром: его поток ввода – это команды и их данные. Однако *sed* и *grep* – типичные фильтры, имеющие собственный командный язык.



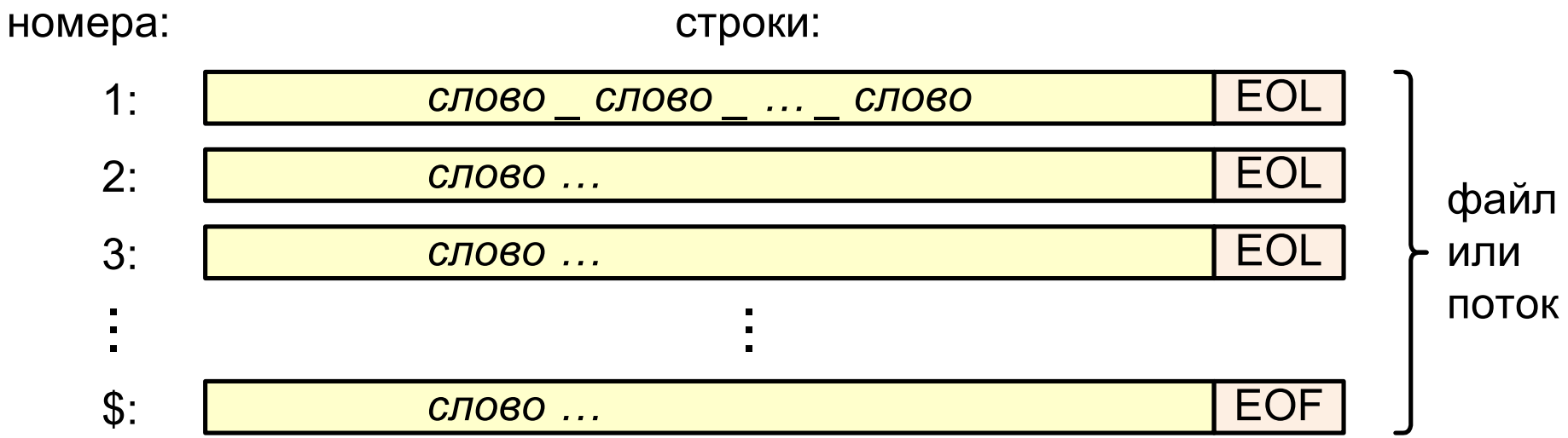
Выполнение редактора *ed*

5.2.2 Представление данных *ed*, *vi*, *sed*, *grep*

Редакторы *ed*, *vi* и *sed* загружают текстовый ***документ*** в буфер и выполняют над ним операции согласно ***командам***, поступающим из потока ввода (*stdin*) или файла. Текст в течение всего сеанса работы остается в буфере, запись в файл только по команде, вывод – тоже (для *ed* и *sed*).

В зависимости от конкретной утилиты и способа ее выполнения, источником документа может служить как файл, так и входной поток (*stdin*).

В любом случае документ структурируется:



Структура текста для *ed*, *vi*, *sed*, *grep*

Входной документ представляется как последовательность строк произвольной длины. Признаком конца строки служит символ *EOL* (стандартно для Unix – *LF*, «**\n**»), признаком конца документа – окончание дискового файла либо прекращение («обрыв») входного потока.

Строки нумеруются начиная с `1`, и к каждой строке можно обратиться по ее номеру (индексу).
Для обозначения последней строки потока или файла используется индекс «`$`».

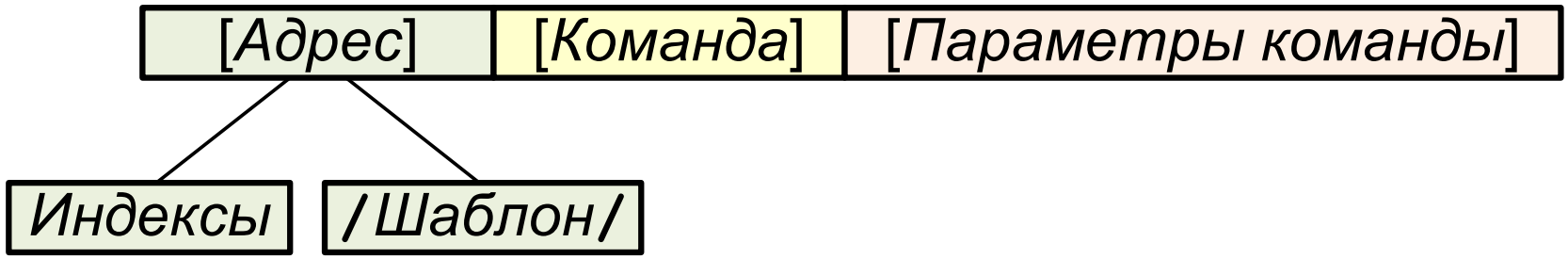
Строка считается состоящей из слов – подстрок, разделенных «пробельными» символами (пробелами или табуляциями). Нумерация слов и обращение к отдельным словам в строке по их номерам (индексам) не предполагаются. Однако на слова (и вообще любые подстроки в составе строки) можно сослаться посредством шаблонов – ***регулярных выражений*** (см. ниже)

Каждая строка обрабатывается независимо от предыдущих и последующих строк. Измененная строка возвращается в буфер и замещает прежнюю ее версию. Следующая команда будет применяться уже к модифицированному содержимому буфера.

5.2.3 Команды *ed* (*ex*), *vi* (*vim*) и *sed*

Множества команд *ed* (*ex*), *vi* (*vim*) и *sed* частично пересекаются, но не совпадают полностью, и могут выполняться с отличиями, в соответствии со спецификой конкретной программы.

В общем случае команда состоит из трех частей, каждая из которых не является обязательной (действуют умолчания). Естественно, все три части одновременно отсутствовать не могут.



Команда *ed*, *vi*, *sed*

Адрес – задает множество строк, к которым будет применена команда. Два способа адресации:

Адресация по ***индексам*** (номерам) строк – явное указание одной или нескольких строк:
N – одна *N*-я строка (в том числе `1`, но не `0`)
M,N – строки от *M* до *N*
N~i – каждая *i*-я по счету строка (т.е. с шагом *i*), начиная с *N*-й (если *N*=`0`, то каждая *i*-я, начиная с *i*-й)

Специальные обозначения:
`$` – последняя строка
`.` – текущая строка

По умолчанию интервал отсчитывается от текущей строки:
«***M,***» – эквивалентно «***M,.***», «***,N***» – эквивалентно «***.,N***»

Относительная адресация (работает в *vi*/*vim* и в GNU *sed*): «**…-*N***» или «**…+*N***» – N-я строка соответственно до или после строки, заданной индексом, но не включая саму эту строку.

Например:

Адрес	**Соответствие**
`1`	только одна первая строка
`1,10`	первые 10 строк буфера
`.,$`	строки от текущей до конца буфера
`.+1`	одна строка, следующая после текущей (в *vi*/*vim*)
`$-9,$`	последние 10 строк (в *vi*/*vim*)
`1~2`	все нечетные строки: 1-я, 3-я, 5-я и т.д.
`0~2`	все четные строки: 2-я, 4-я, 6-я и т.д.

Ассоциативная адресация – посредством шаблона (регулярного выражения): команда будет применена к тем строкам, которые соответствуют (см. ниже) шаблону. Вся адресная часть интерпретируется как шаблон, признаком это служит заключение его в «слэши»: **`/ … /`**.

Относительная адресация («**+**») и адресация с шагом («**~**») работают также и в сочетании с шаблонами (GNU *sed*).

Например:

Адрес	**Соответствие**
`/#include/`	все строки C-программы со ссылками на .h-файлы
`/^$/`	все пустые строки
`/^[ \t]*$/`	все пустые или пробельные строки
`/[^ \t]+/`	все непустые непробельные строки

По умолчанию команды действуют на все содержимое буфера, т.е. адресная часть подразумевается «**`1,$`**»

Команда – единственный символ, определяющий выполняемое действие, например: **`w`** – *W*rite, **`a`** – *A*ppend, **`t`** – *T*ransfert, и так далее.
По умолчанию принимается команда **`p`** – *P*rint.
В некоторых случаях несколько команд могут объединяться, например **`wq`** – *W*rite-and-*Q*uit.

Параметры команды – зависят от конкретной команды.

Некоторые основные команды:

`q` – *Q*uit – выход из редактора (в интерактивном режиме)
`q!` – принудительный выход без сохранения буфера

`r` – *R*ead – чтение из файла в буфер в текущую позицию
`w` – *W*rite – запись содержимого буфера в файл

`a` – *A*ppend – добавление нового содержимого после текущей позиции
`i` – *I*nsert – вставка нового содержимого перед (начиная с) текущей позиции
`c` – *C*hange (предположительно) – замещение содержимого
(в интерактивном режиме *vi*/*vim* эти три команды действуют почти одинаково)

`p` – *P*rint – вывод содержимого буфера (в *vi*/*vim* – переход к началу адресуемого диапазона)

`m` – *M*ove –перенос строк (не работает в *sed*)

`t` – *T*ransfert – копирование строк (не работает в *sed*)

`d` – *D*elete – удаление строк в буфере

`u` – *U*ndo – отмена последней команды

Например:

Команда	**Эффект**
`1,10p`	вывод первых 10 строк (аналогично фильтру *head*)
`2m3`	обмен местами 2-й и 3-й строк (вставка после строки, указанной как параметр)
`3m2`	видимого эффекта нет (перенос на свое же место)
`.t.`	дублирование текущей строки
`$d`	удаление последней строки буфера (потока)
`/^$/d`	удаление пустых строк

`y` – табличная подстановка символов во всей строке (работает в *sed*).

Формат:

`y/`*`символы`*`/`*`замены`*`/`

Например:

Команда	**Эффект**
`y/abcdef/ABCDEF/`	замена нижнего регистра букв на верхний
`y/alnos/@1#0$/`	замена букв на цифры и знаки схожего начертания

`s` – подстановка/замена подстрок по шаблонам

Формат:

`s/шаблон_подстроки/шаблон_замены/опции`

Среди опций часто используются:
`g` – «глобальность» (замена всех встреченных соответствий, иначе только первого найденного)
`i` – регистронечувствительность при поиске соответствий
`c` – запрос подтверждений для каждой замены, и т.д.

Например:

Команда	**Эффект**
`s/for/4/gi`	замена всех сочетаний «for», «For» и т.п. на «4»

Более подробно эта команда будет рассмотрена вместе с соответствующими регулярными выражениями (см. ниже).

Выполнение команды ОС/shell, в т.ч. составной (в *vi*/*vim*):

```
! команда
```

Например, загрузка файла с исходным текстом в *vim*, редактирование, сохранение, компиляция и выполнение написанной программы:

```
vim myprog.cpp  # далее работа идет внутри vim
... # просмотр и редактирование исходного текста
:w
:! gcc myprog.cpp -o myprog
:! ./myprog
```

Явная запись файла здесь необходима, иначе его содержимое не меняется, и все изменения остаются лишь в буфере.

Команды, выполняемые в «расширенном» режиме *vi*/*vim*, обычно предваряют двоеточием – это команда перехода в Ex-mode, например «**`:!`**»,«**`:w`**», «**`:s`**», «**`:u`**» и т.п. (см. ниже)

Для упрощения большинство примеров приведено без адресной части команд.

5.2.3 Визуальный редактор *vi* (*vim*)

Многорежимный, командно-ориентированный «визуальный» текстовый редактор *vi* (*VI*sual).

Особенность (наследство **`ed`**): загрузка документа в буфер и далее работа с содержимым буфера, сохранение только принудительное (по команде). Эффективно работает с документами очень большого размера.

«Усовершенствованная» версия – *vim* (*VI* i*M*proved), имеет ряд улучшений, включая подсветку синтаксиса редактируемого текста (распознаются исходные тексты на различных языках программирования), собственный shell-образный скриптовый язык и т.д. Более удобно работает с современными консолями.

Режимы:

Командный (основной) – навигация по тексту и некоторые операции редактирования. Возврат в этот режим из других – клавиша **[ESC]**

Режим интерактивного ***редактирования*** – Добавление, вставка, замещение текста (можно считать тремя отдельными режимами) – собственно интерактивное редактирование.
Переход в режим – команды «**a**», «**i**», «**c**».
Выход из режима – клавиша [**ESC**].

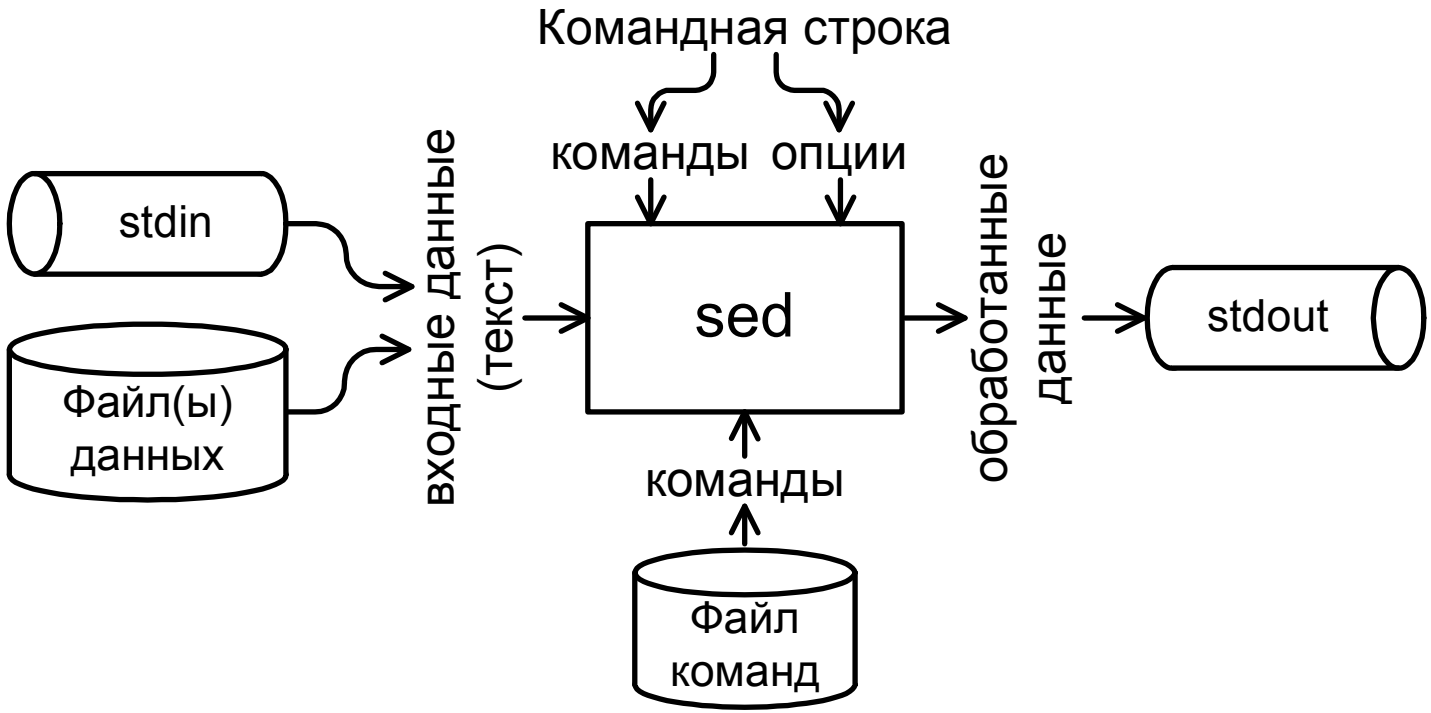
«***Расширенный***» командный режим (*Ex*-mode)
Переход в режим – команда «**:**».
Выход из режима – клавиша [**ESC**] или по завершении команды.

Большинство ранее рассмотренных команд «массового» редактирования документа, работают в «расширенном» режиме.

5.2.4 Потоковый редактор *sed*

«Потоковый редактор» *sed* (*S*tream *ED*itor) – ***неинтерактивное*** средство обработки текстов. Можно рассматривать как сложный универсальный ***фильтр*** для разнообразной обработки текстовых данных, имеющий собственный специализированный командный язык с широким использованием ***регулярных выражений*** (см. ниже).

Основной сценарий использования – применение к входному потоку текстовых данных одной или нескольких команд, но может работать и с входным файлом или множеством файлов (аналогично большинству программ-фильтров). Результаты всегда передаются в поток и при необходимости могут быть перенаправлены.



Выполнение потокового редактора *sed*

Типичная командная строка для sed:

`sed -`*`опции`* `'`*`команда`*`' [`*`файл файл`* `...]`

Экранирование команд *sed* необходимо, так как среди них могут оказаться метасимволы *shell* (или иной «внешней» программы), которые приходится «прятать», причем часто используются одинарные кавычки – безусловное экранирование, без раскрытия переменных *shell*.

Наиболее часто используемые опции:

`-n` – «тихий» режим, подавление вывода в *stdout* всех строк потока после обработки; останутся только результаты явных команд печати «**`p`**».

`-e` – «расширенный» синтаксис команд: возможность задать в командной строке несколько команд *sed*, разделяя их точками с запятой.

`-f` – выполнять команды из заданного файла

Примеры:

Вывод 10 первых строк потока (аналог фильтра **head**):

```
sed -n '1,10p'
```

Вывод 10 первых непустых строк файла (неоптимальная реализация):

```
sed '/^$/d' myfile.txt | sed -n '1,10p'
```

Сочетание опций и расширенный синтаксис, но эффект не вполне соответствует желаемому:

```
sed -ne '/^$/d;1,10p' myfile.txt
```

Выполнение сложной последовательности команд из файла-скрипта, результат записан в файл:

```
sed -f myscript.sed myfile.txt > myresult.txt
```

Фрагмент скрипта shell с переменной в команде **sed** (здесь «**${cnt}**» содержит метасимволы shell и будет преобразовано до передачи в **sed**):

```
cnt=10
sed -n "1,${cnt}p" ...
```

Основная часть команд **sed** уже рассмотрена выше. Примеры применения команды «**s**» будут рассмотрены вместе с соответствующими регулярными выражениями.

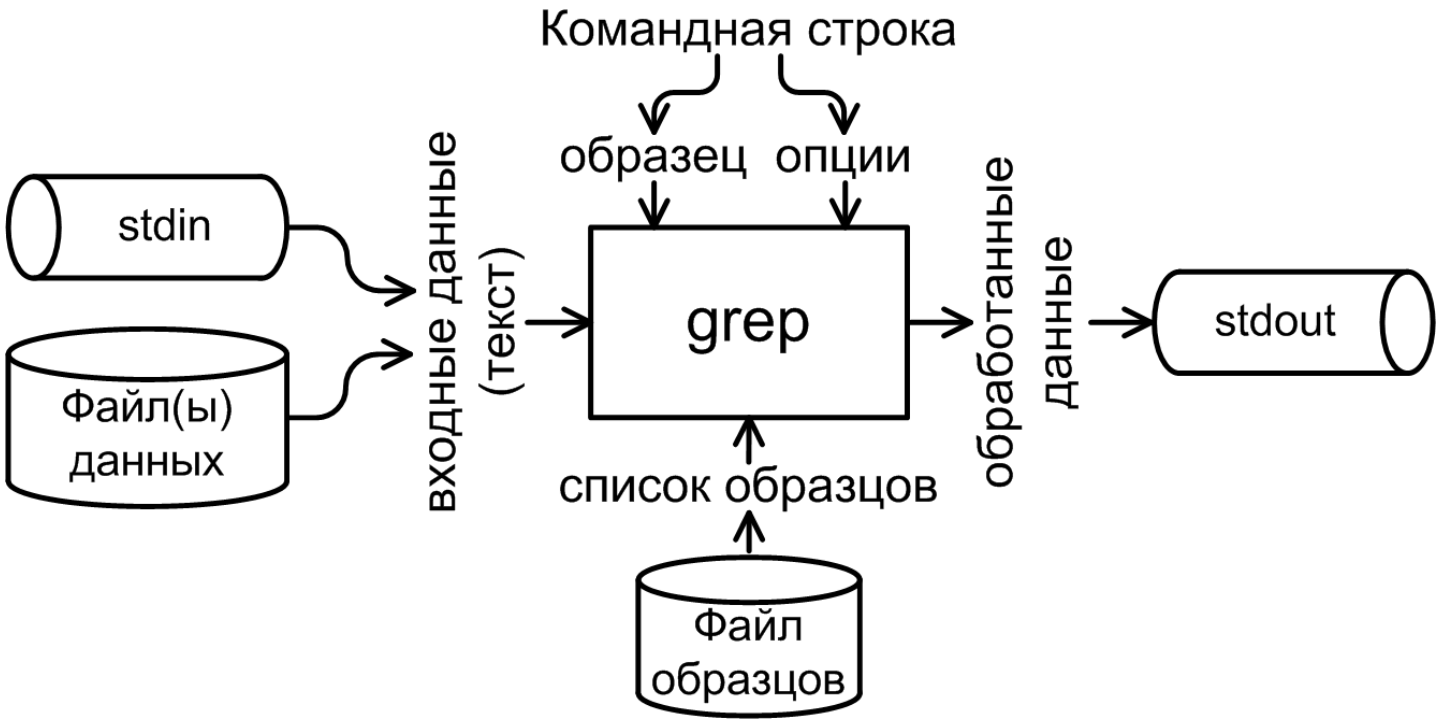
5.2.5 Утилита поиска строк grep

Утилита (точнее, семейство утилит) *grep*[2] – программа-фильтр для поиска по образцу строк в файлах и/или потоках.
Одна из наиболее часто используемых программ.

По умолчанию *grep* выводит строки, в которых обнаружено соответствие с заданным образцом поиска.
Образец рассматривается как регулярное выражение (обычно, иначе см. ниже).

Таким образом, *grep* выполняет более узкий по сравнению с *sed* набор функций и поэтому не нуждается в отдельном командном языке, но одновременно он имеет дополнительные возможности именно поиска (фильтрации).

[2] Считается, что наименование *grep* образовано из описания ее функций с привлечением синтаксиса команд *ed*: «*g*lobally search for a *r*egular *e*xpression, *p*rint matching lines» – *g*/*re*/*p*.



Выполнение фильтра *grep*

Типичная командная строка для *grep*:

`grep -`*`опции`* `'`*`образец`*`'` `[`*`файл файл`* `...]`

Существует целое семейство программ *grep*, например:

egrep – расширенный синтаксис регулярных выражений

fgrep – поиск совпадений по списку «фиксированных» образцов (не регулярных выражений!)

Использование двух этих программ считается нежелательным, рекомендуется эмулировать их опциями *grep*

rgrep – рекурсивный поиск в файлах указанного директория и его поддиректориев (эмулируется опциями *grep*)

pcregrep – Perl-совместимый диалект регулярных выражений

agrep – поиск «приблизительных» соответствий образцу (*a*pproximate *grep*)

pgrep – поиск соответствий образцу среди имен процессов в системе (т.е. очень специализированный *grep*)

Далее рассматривается «стандартный» *grep* (GNU-версия).

Некоторые опции, влияющие на поведение *grep*:

Опция	**Эффект**
`-E`, **`-B`**, **`-P`**	использовать синтаксис регулярных выражений: расширенный, базовый, Perl-совместимый
`-F`	поиск по списку фиксированных образцов
`-f`	использовать список образцов из файла
`-r`	рекурсивный поиск в файлах директория
`-i`	искать соответствия без учета регистра букв
`-v`	инверсия соответствий (искать несоответствия)
`-w`, **`-x`**	искать соответствия целым словам или строкам
`-o`	вывод только найденных подстрок, не строк целиком

Опция	**Эффект**
`-c`	вывод только счетчика соответствий
`-q`	подавление всего вывода, только вернуть код результативности (**0** – соответствие найдено)

Вариант вызова потокового редактора *sed* – функциональный аналог *grep* (естественно, с точностью до диалекта регулярных выражений и без поддержки специфических опций *grep*):

`sed -n '/`*`шаблон_фильтрации`*`/p'` …

5.3 Регулярные выражения (Regular Expression)

5.3.1 Основные сведения о регулярных выражениях

В общем смысле, ***регулярное выражение*** (***regular expression*** или ***regexp***) – последовательность символов и ***метасимволов*** (специальных символов), которую может интерпретировать (разобрать) ***конечный автомат***. И наоборот, конечный автомат может быть определен через возможность обработки им регулярных выражений.

В более узком смысле, регулярное выражение – способ описать множество текстовых строк через соответствие данному регулярному выражению. Аналогично, регулярное выражение может быть использовано для порождения множества соответствующих ему текстовых строк.

Регулярные выражения состоят из символов и метасимволов. Символы регулярного выражения соответствуют самим себе в строке данных, метасимволы – управляют интерпретацией отдельных символов и всего выражения.

В нашем контексте базовая операция с использованием регулярный выражений – проверка на ***соответствие***.
В простейшем случае обычный символ или строка обычных символов тоже являются регулярным выражением, но они соответствуют сами себе. Наличие метасимволов делает соответствие более сложным и многообразным.

Существует множество «диалектов» регулярных выражений, различающихся в том числе и отношением к метасимволам.

5.3.2 Группы метасимволов

Экранирование

Символ «\» (обратный слэш) – «экран»: экранирует (снимает) специальный смысл у метасимвола и включает его у обычного символа (если он предусмотрен).

«Якоря» (Anchors)

Обеспечивают позиционирование в строке:

Метасимвол	Соответствие
^	начало строки
$	конец строки
\<	начало слова (поддержка зависит от диалекта)
\>	конец слова (поддержка зависит от диалекта)

Символьные классы

Описывают соответствие одному символу строки:

Метасимвол	Соответствие
`[ … ]`	перечисление допустимых символов
`[ …-… ]`	диапазон допустимых символов
`[^ … ]`	инверсия символьного класса

Метасимвол диапазона «`-`» в начале перечисления, символ инверсии «`^`» не в начале перечисления, а также прочие метасимволы в перечислении интерпретируются как обычные символы.

Также предусмотрен «универсальный» метасимвол-заменитель (можно считать предопределенным символьным классом):

`.`	один произвольный символ

Некоторые диалекты включают более сложные предопределенные классы, например **`{:alpha:}`** – любая буква.

Примеры описаний символьных классов:

Класс	Соответствие
`[ \t]`	«пробельный» символ – пробел или табуляция
`[^ \t]`	непробельный символ – не «␣» и не «`\t`»
`[0-9]`	десятичная цифра
`[a-ZA-Z]`	буква латиницы
`[0-9A-Fa-f]`	шестнадцатиричная цифра
`[-+*/%^]`	знак арифметического действия
`[-.,:;!?]`	пунктуационный знак

Квантификаторы (Quantifiers)

Определяют соответствие заданному количеству повторов предшествующей подстроки или символа[3]:

Метасимвол	Соответствие
`\?`	необязательный символ (или группа)
`\+`	один или более повторов
`*`	ноль, один или более повторов
`\{`*n*`\}`	точное число повторов *n*
`\{`*n*`,\}`	число повторов не менее *n*
`\{`*m*`,`*n*`\}`	число повторов от *m* до *n*

[3] Поведение квантификаторов может различаться в зависимости от диалекта: «ленивое» или «жадное», воздействие на один символ или подстроку начиная от разделителя (например, символа группировки или предыдущего квантификатора), и т.д.

Например:

Шаблон	Соответствие
`^[ \t]*`	необязательные стартовые пробельные символы
`[0-1]\{4\}`	4-значное десятичное число (целое)
`ABC\{3\}`	тройной повтор подстроки «ABC»

Вариант (логическое «ИЛИ»)

Отписывает соответствие одному из образцов (не обязательно состоящих из единственного символа:

Метасимвол	Соответствие
шаблон1*|*шаблон2	«***шаблон1***» или «***шаблон2***»

Например:

Шаблон	Соответствие
[Mm]arch|[Aa]pril|[Mm]ay	любой из весенних месяцев

Группировка

Объединяет ряд символов и метасимволов в группу, с которой можно обращаться как с единым образцом.

Метасимвол	Эффект
`\(`***`подстрока`***`\)`	«***подстрока***» рассматривается как группа

Это позволяет, например, гарантированно применить квантификатор к подстроке или сформировать «составной» шаблон:

Шаблон	Соответствие
`[01]\{1,4\}\(\.\?[01]\{4\}\)*`	запись двоичного числа с необязательным делением его точками на тетрады (пример имеет дефекты)

Группировка также существенно расширяет возможности обратных ссылок (см. ниже).

Обратные ссылки

Во втором аргументе команды «**s**» (т.е. в редакторах, но не в *grep*) позволяют обратиться к ранее найденным соответствиям, если они были оформлены как группы (см. выше), или к всей той части строки, которая была найдена соответствующей:

Метасимвол	Соответствие
`\1`, `\2`, …, `\9`	найденные группы с 1-й по 9-ю
`&`	вся соответствующая шаблону (первый аргумент) часть строки

Важно, что для каждого найденного соответствия в каждой обрабатываемой строке каждый раз подставляется новое актуальное значение.

Пример: команда, преобразующая даты из формата «mm/dd/yyyy» в «dd.mm.yyyy»:

```
s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)\
/\2\.\1\.\3/g
```

(Пример упрощен: в частности, не учитываются запись дат без разделителей между ними, пропуск незначащих нулей, «сокращенный» двузначный номер года и т.п.)

5.3.3 Правила поведения регулярных выражений

Наиболее общие правила, которых придерживается большинство реализаций (но не обязательно все!):

1) Среди найденных альтернативных (пересекающихся) соответствий предпочитается первое, т.е. то, которое началось раньше)

2) Квантификаторы работают «максимально» («жадное» поведение), т.е. накрывают наибольшее возможное число повторов образца. В то же время, «жадность» квантификатора ограничивается «просмотром вперед».

Это надо учитывать при использовании метасимволов-«джокеров» (« . »), обратных ссылок и т.д.

5.3.4 Требования к регулярным выражениям и их оптимизация

1) Точность, надежность, однозначность совпадения.

2) Понятность и управляемость выражений (при том, что синтаксис в целом для чтения неудобен).

3) Эффективность – быстрое схождение к результату.

Оптимизация регулярных выражений и операций с их использованием актуальна в силу того, что их обработка – вообще достаточно сложная и ресурсоемкая задача, требующая просмотра большого дерева вариантов. Основной подход – сокращение этого дерева отбрасыванием заведомо нерезультативных путей и по возможности ранним выявлением наиболее часто встречающихся соответствий.